Sesión 16: Listas (1/2)

Programación 2

Ángel Herranz

2024-2025

Universidad Politécnica de Madrid

En capítulos anteriores

- 👍 Tema 1: Intro a POO
- 👍 Tema 2: Clases y Objetos y TADs y módulos
- 👍 Tema 3: Colecciones con *arrays*
- 🕒 Tema 4: Cadenas simplemente enlazadas

En el capítulo de hoy

- Tema 4: Cadenas simplemente enlazadas
- Tema 5: TAD Listas
- Tema 7: Herencia[1] y Polimorfismo

[1] Hoy sólo *herencia de interface*.

TAD Listas i

Un TAD (tipo abstracto de datos) lo definen

- Un nombre, el nombre del tipo.
- Unas operaciones.
- La semántica de las operaciones.

TAD Listas i

Un TAD (tipo abstracto de datos) lo definen

- Un nombre, el nombre del tipo.
- Unas operaciones.
- La semántica de las operaciones.
- El mismo TAD puede tener múltiples implementaciones concretas: estructuras de datos.

TAD Listas ii

API

`IListStr`

+

`add`, `get`, `size`, `set`, `indexOf`, `remove`, `subList`

TAD Listas ii

API

`IListStr`

+

`add`, `get`, `size`, `set`, `indexOf`, `remove`, `subList`

Semántica en `javadoc`

TAD Listas ii

API

`IListStr`

+

`add`, `get`, `size`, `set`, `indexOf`, `remove`, `subList`

Semántica en `javadoc`

`LinkedListStr`

Cadena simplemente enlazada

`ArrayListStr`

Arrays (redimensionables)

TAD Listas ii

API

`IListStr`

+

`add`, `get`, `size`, `set`, `indexOf`, `remove`, `subList`

Semántica en `javadoc`

`LinkedListStr`

Cadena simplemente enlazada

`ArrayListStr`

Arrays (redimensionables)

API: *interface* `IListStr`

```
public interface IListStr {
  void add(String elem);
  void add(int index, String elem);
  String get(int index);
  int size();
  void set(int index, String elem);
  int indexOf(String elem);
  void remove(int index);
  void remove(String elem);
  IListStr subList(int start, int end);
}
```

¿Qué es `interface`? i

- **`public interface`** `IListStr`
- *Clase* sin implementación de los métodos
- Perfecto para representar TADS:

  nombre del *tipo* + nombres de operaciones
- Es un tipo (declaración de variables y métodos)
- No se puede instanciar
- Otras clases pueden implementar el *interface*

¿Qué es `interface`? ii

- Compilar este programa principal:

```
public class Sesión17 {
  public static void main(String[] args) {
    IListStr l = new IListStr();
  }
}
```

- Entonces ¿cómo podemos crear instancias de `IListStr`?

¿Qué es `interface`? iii

- Crear una clase que implemente el *interface*:

```
public class LinkedListStr implements IListStr {
  public void add(int index, String elem) {
  }
  ...
}
```

- Y ahora...

```
IListStr l = new LinkedListStr();
```

¿Qué es `interface`? iv

- Herencia de interfaz:

```
class LinkedListStr implements IListStr
```

- La clase `LinkedListStr` implementa `IListStr` y por lo tanto el programador está obligado a implementar todas las operaciones
- Una instancia de `LinkedListStr` puede guardarse en variables de tipo `IListStr`
- Sobre variable de tipo `IListStr` se puede aplicar todos los métodos declarados en el *interface*

¿Qué es `interface`? v

- No es necesario **`public`** porque todo es público en un *interface*
- Una misma clase puede implementar varios *interfaces*:

  Java tiene herencia múltiple de interfaz
- Los *interfaces* se usan para hacer buenos diseños ocultando implementaciones y exigiendo APIs
- Hazte preguntas e intenta contestarlas (ej. ¿puedo declarar atributos? ¿qué pasa con ellos en la clase que implementa? ¿private?)

Semántica: ejemplo en `javadoc`

Continuamos con el TAD Listas: semántica

```
/**
 * Quita de la lista el elemento en la posicion {@code index}.
 *
 * @pre. {@code index} mayor o igual que 0 y menor que {@code size()}
 * @post. {@code this} después de ejecutar representa una lista como
 *        {@code this} antes de ejecutar eliminando el elemento que
 *        ocupaba la posición indicada en el parámetro {@code index}
 * @param index el índice del elemento que se va a eliminar
 * @throws RuntimeException si no se cumple la precondición
 */
void remove(int index);
```

Precondición y Postcondición (Def.)

Precondición

Predicado que habla del estado de mi programa (incluídos los parámetros de llamada)

Postcondición

Predicado que relaciona el estado después de que la operación se haya ejecutado con el estado antes de que se ejecute

Precondición y Postcondición (Def.)

Precondición

Predicado que habla del estado de mi programa (incluídos los parámetros de llamada)

$$0 \leq index \wedge index < size()$$

Postcondición

Predicado que relaciona el estado después de que la operación se haya ejecutado con el estado antes de que se ejecute

Precondición y Postcondición (Def.)

Precondición

Predicado que habla del estado de mi programa (incluídos los parámetros de llamada)

$$0 \leq index \wedge index < size()$$

Postcondición

Predicado que relaciona el estado después de que la operación se haya ejecutado con el estado antes de que se ejecute

"`this` despues de ejecutar . . ."

Semántica pre-post i

La especificación pre-post

es un contrato

"Yo (el programador que implementa) garantizo que se cumple la postcondición, siempre y cuando tú (el programador que usa) respetes la precondición."

Semántica pre-post ii

- ¿Y si el programador que usa la operación no respeta la precondición?

  Ej. `lista.get(-2)`

- ¿Qué pasará?

Semántica pre-post ii

- ¿Y si el programador que usa la operación no respeta la precondición?

  Ej. `lista.get(-2)`

- ¿Qué pasará? Se rompe el contrato:

Semántica pre-post ii

- ¿Y si el programador que usa la operación no respeta la precondición?

  Ej. `lista.get(-2)`

- ¿Qué pasará? Se rompe el contrato:
  1. La postcondición **no está garantizada**
  2. Esto implica que **no se sabe lo que va a pasar**
  3. En el mejor de los casos el programa se rompe o eleva una **excepción** (programación defensiva)
  4. En **el peor de los casos los datos se corrompen** (¡no sólo los del programa!)

Javadocumentar con pre-post

Javadocumentar el resto del *interface* usando especificaciones pre-post asumiendo que hacemos programación defensiva

Implementar `LinkedListStr` i

- Crear el fichero `IListStr.java` con el *interface*
- Crear el fichero `NodeStr.java`

```
class NodeStr {
  String elem;
  NodeStr next;
}
```

- Implementar `LinkedListStr.java`

```
public class LinkedListStr implements ListStr {
  private NodeStr first;
  ...
}
```

Implementar `LinkedListStr` ii

1. *Test-first*: implementación *vacía* de la clase `LinkedListStr` + `ListStrTest.java`
2. ⚠ A la hora de implementar `LinkedListStr` tendremos que adaptar el API *funcional* de las sesiones anteriores a un API más OO:

```
static String obtener(NodoStr cadena, int i)
```

vs.

```
String get(int i)
```

Implementar `LinkedListStr` iii

Puede ayudar implementar primero `toString` para poder ver la lista entre prueba y prueba. Por ejemplo:

```
IListStr l = new LinkedListStr();
System.err.println(l);
assert l.size() == 0;
l.add(0, "Hola");
System.err.println(l);
assert l.size() == 1;
assert "Hola".equals(l.get(0));
```